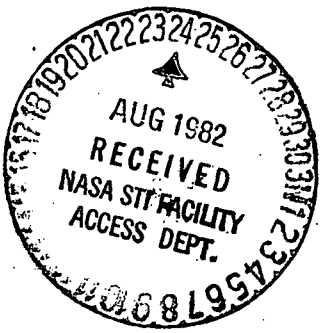
# The Implementation and Use of Ada On Distributed Systems

# With High Reliability Requirements

Semi-Annual Progress Report




John C. Knight

Paul F Reynolds

Department of Applied Mathematics and Computer Science

University of Virginia

Charlottesville

Virginia, 22901

August 1982

## 1. Introduction

The purpose of this grant is to investigate the use and implementation of Ada (a trade mark of the US Dept. of Defense) in distributed environments in which the hardware components are assumed to be unreliable. In particular, we are concerned with the possibility that a distributed system may be programmed entirely in Ada so that the individual tasks of the system are unconcerned with which processor they are executing on, and that failures may occur in the underlying hardware.

Over the next decade, it is expected that many aerospace systems will use Ada as the primary implementation language. This is a logical choice because the language has been designed for embedded systems. Also, Ada has received such great care in its design and implementation that it is unlikely that there will be any practical alternative in selecting a programming language for embedded software.

The reduced cost of computer hardware and the expected advantages of distributed processing (for example, increased reliability through redundancy and greater flexibility) indicate that many aerospace computer systems will be distributed. The use of Ada and distributed systems seems like a good combination for advanced aerospace embedded systems.

Our work under this grant up to this point indicates that the situation is not as good as expected. There seem to be numerous aspects of the language which make its use on a distributed system somewhat difficult. The issues are not raised directly from efforts to implement the language but from the desire to be able to recover, reconfigure, and provide continued service in the presence of hardware failure.

Our work so far has consisted of:

(1) A formal definition of the Ada tasking semantics.

(2) An examination of the language (July 1982, version 9) to thoroughly understand the various features.

(3) The generation of a model of the assumed underlying hardware system and the failures of that system which will be tolerated.

(4) An examination of the Ada language with regard to its use and implementation in the assumed environment.

(5) The design of an experimental system that can be used to investigate the algorithms we expect to develop for the implementation of Ada.

Each of these topics are discussed in the sections below.

## 2. Formal Semantic Definition of Ada

In order to be able to implement a language, it is imperative that a precise definition of the language semantics be available. Semantic definitions of programming languages are relatively uncommon because existing methods for semantic definition are difficult to use and in some ways inadequate. A semantic definition of Ada has been prepared for the Ada Joint Program Office (AJPO) using denotational semantics. This definition is quite difficult to read but its biggest problem is that the tasking and exception semantics of Ada are totally absent from the definition. The reason is that denotational semantic methods are not sufficiently powerful to describe tasking.

Previous work at the College of William and Mary produced a semantic definition using H-graph semantics. Since that report, the Ada language has changed somewhat and the H-graph definition methodology has been revised considerably. The new definition which we have undertaken is a

revision of the earlier work at William and Mary using the latest versions of both Ada and the H-graph method. The current version of the definition is attached as appendix 1.

## 3. Examination of the Language - General

As a part of the activities under this grant, we participated as a volunteer review group for the July 1982 Ada definition. This effort was coordinated by AdaTEC as an attempt to generate comments from the United States about the revised language before the final version (which would be used for the ANSI canvass) was printed. We found the revised language definition document to be very different from the July 1980 version and chose to put all of our effort into reviewing the tasking definition thoroughly rather than review the entire language definition superficially.

The result of our review was a set of 35 questions which were discussed at the meeting of the volunteer reviewers at the AdaTEC conference in Boston (June 1982). Our comments were well received and many were found to have substantial content. These were to be passed on to the Ada design team for consideration. A brief examination of the July 1982 Ada reference manual (denoted version 16) indicates that our comments were either not received in time or were not acted upon since most of the language difficulties still exist. A copy of our questions is attached as appendix 2.

In general our concerns about Ada are to do with time. Some examples are:

(1) The conditional entry call does not define "immediately" and so we have no way of knowing how the call is supposed to be implemented.

There are several ways which are entirely different based on the current language reference manual definition.

(2) The timed entry call does not define when the time for the call is to begin. There are again several different interpretations. Worse however, is the fact that the timed entry call does not provide a useful facility for the programmer in its current form given any definition. The problem which it should address is the need to be able to time out easily once a rendezvous has begun rather that before it has begun.

These issues demonstrate why a formal definition of Ada is so important. We cannot decide exactly what is supposed to be implemented given the current language definition which is in English and therefore ambiguous, imprecise, and incomplete. Unfortunately, the issues with the conditional and timed entry calls, and with other language elements, are actually much more serious in a distributed system.

## 4. Underlying Hardware Model

Initially, we assume that communication between processors on a distributed system will be implemented using layers of software that conform for the most part to the ISO standard seven layer Reference Model. The hardware topology that is used for a distributed system need have very little impact on the programming of the system at the application layer level. In principle, provided the implementation knows how tasks are distributed to processors and how communication is to be achieved, the various tasks can synchronize and communicate at will with no knowledge of their location.

To discuss implementation and recovery in the context here, we found it necessary to have an underlying hardware model. Our model assumes

that a set of processors are connected to some sort of communications bus system which we do not define. The bus system could be a ring, multiple rings, a crossbar switch, etc. Peripheral equipment such as sensors and actuators are also assumed connected to the bus system, and the connection is assumed to be provided by a microprocessor dedicated to the interface.

The kinds of hardware failure that we are concerned with are not addressed by the ISO protocol. The ISO protocol is concerned with minor communications failures such as dropped bits caused by noise, etc. Also, situations such as a processor "slowing down" or incorrectly computing results are not of interest here (though they are important never the less). We assume that such events are taken care of by hardware checking within the processor or similar. The only class of faults not dealt with elsewhere is the sudden total loss of a processor or bus and these are the difficulties we will attempt to deal with.

We feel that this hardware model and associated failure model is directly relevant to avionics systems but also seem very relevant to spacecraft systems. Sophisticated unmanned spacecraft frequently make extensive use of computers (e.g. Galileo) but are unable to pay the weight and power costs of extensive redundancy (such as in quad redundancy or SIFT). Reconfigurable distributed systems designed to cope with processor or bus failure is an attractive alternative. If the design includes higher processing power than is absolutely needed, and tasks exist which are not essential to mission success, then some loss of hardware followed by reconfiguration may allow the mission to continue successfully.

## 5. Ada on Unreliable Distributed Systems

We are examining the Ada tasking facilities to see which ones can be supported and which ones have to be abandoned in the programming of distributed systems on unreliable hardware. The restrictions have to be imposed because of the need to reconfigure following hardware failure.

A key problem area is the association of processors with tasks nested within other tasks. Failure of a processor containing a parent task whose children are running on another processor leaves the children as "orphans". Such orphan tasks present a difficult problem for reconfiguration because if the parent is to be restarted on a different processor, the children will be recreated as the parent is elaborated.

Another difficult problem area is detection of hardware failure. The hardware may not know that it has failed. The only proposal that seems feasible at the moment is the use of software heartbeats within each processor that are software monitored in other processors.

## 6. A System For Experimentation

The confirmation that algorithms developed for use and implementation of Ada are valid is best achieved by experimentation. We propose to construct an experimentation facility for distributed systems using our departmental VAX 11/780 computer. In the overall design, we intend to have the VAX function as a programmable switch to which several smaller computers will be connected. The VAX will be programmed to simulate any bus structure that is desired for connecting the smaller computers. In addition, the VAX can systematically and repeatedly inject faults into the simulated communications and monitor the handling of the faults. In this way bus and

processor failure can be simulated easily. At present, we are at the early design stage of the overall system and the switch software.

## 7. Remaining Grant Activities

During the remaining grant period, we are going to complete our study of the use of the language so as to provide a complete set of guidelines for writing Ada programs in the context of interest. At present, we anticipate that the list of restrictions will be quite substantial.

Once we have completed that study, we are going to look closely at implementation. This effort may be easier than at first thought. If software heartbeats are used, the responsibility for error detection and management of the reconfiguration will be at the level of the application software layer. The implementation can be fairly standard except for those primitives used to provide reconfiguration services.

If software heartbeats are not used, the implementation will be much more involved because it must provide all the major aspects of the required fault tolerance.

# Appendix 1

An operational model of Ada tasking has been devoloped using an H_graph notation developed in. 'H_Graph Semantics'. T.W.Pratt. Technical Report Department of Applied Mathematics and Computer Science. University of Virginia.Sept 1981.

In the model each task is assummed to be running on a separate processor: communication between processes (and hence between processors) is by remote calls to kernel procedures. The run time state is described by an H_graph grammar (A). Every instruction will ultimately be defined as a transformation of the run time state.

The transform COMPILE (B) translates an Ada text into an intermediate form consisting of a list of declarations and a list of executable statements (ie transformations of the run time state). Execution of the intermediate form proceeds in two steps. First elaboration of the declaration list. and second. execution of the statement list.

An example of Ada text (C) and the intermediate form obtained by the application of COMPILE (D) is included.

A.

```
network ::=
    [ [NETWORK]
        -<node_id>->    network_node
       {-<node_id>->    network_node }
    ]

network_node ::=
    [ [NETWORK_NODE]
        -name-> [ <node_id> ]
        -processor->    processor
        -communications_interface-> [ [COMM]
                                                    -proc_export->    proc_info
                                    -proc_import_queue-> *piq: QUEUE( proc_info )
                                  ]
       -kernel_proc_code-> [ [KPCODE]
                                -<kproc_name>->    code
                                {-<kproc_name>->    code }
                           ]
       -process->    process
    ]

processor ::=
    [   [PROCESSOR]
        -next_instruction->    instruction_pointer
        -timer-> [-[TIMER_HARDWARE ]
```

```
                        -set-> [ <timer_status> ]
                        -delay-> [ <time> ]
                        -transfer_address->   instruction_pointer
                    ]
        -flags-> [ [FLAGS]
                        -user_pgm_suspended-> [ <boolean> ]
                        -inhibit_timer-> [ <boolean> ]
                -inhibit_abort-> [ <boolean> ]
                -inhibit_exception-> [ <boolean> ]
                -check_immediate_rendezvous-> [ <boolean> ]
            ]

        -network_node-> [ network_node ]
        -kproc_info-> proc_info
    ]

instruction_pointer ::= [ [IP]
                                -instruction-> [ code_node ]
                                -code_block->   [ code ]
                            ]

proc_info ::=
    [   [ PROC_INFO ]
        -to-> [ <node_id> ]
        -from-> [ <node_id> ]
        -kproc_name-> [ <name> ]
```

```
        -parameters-> KEYED_LIST( <integer>,arb_node )
    ]

code ::= LIST( code_node )

code_node ::= instruction_node | branch_node | LIST( code_node )

instruction_node ::= [ [INSTRUCTION_NODE]
                        -transform-> [ <transform_id> ]
                        -arguments-> KEYED_LIST( <integer>,arb_node )
                     ]

branch_node ::= [ [BRANCH_NODE]
                    -condition-> function_node
                    -alternatives-> KEYED_LIST( <integer>,code )
                  ]

function_node ::= [ [FUNCTION_NODE]
                      -function-> [ <function_id> ]
                      -arguments-> KEYED_LIST( <integer>,[ arb_atom ]
                      -result-> [ arb_atom ]
                    ]

process ::= [ [PROCESS]
              -process_object->   process_object
```

```
                -proc_import_queue-> **piq
                ]

process_object ::= [ [PROCESS_OBJECT]
                        -next_instruction-> . instruction_pointer
                        -activation_record_stack-> STACK( activation_record )
                        -load_module-> [ load_module ]
                    ]

activation_record ::= task_activation_record | subprogram_activation_record | package activation record


task_activation_record ::=
    [ [TAR]
        -user_data->    user_data
        -system_data->    system_data
        -elaboration_data-> elaboration_data
    ]

user_data ::=
                [ [USER_DATA]
                    -locals-> --allocated by elaboration
                    -non_locals-> KEYED_LIST( <name>,[<nesting_level>] )
                ]

system_data ::=
    [   [SYSTEM DATA]
```

```
    -my_phone_#-> processing_unit
    -state-> [ <state> ]
    -context->      [ [CONTEXT]
                    -ref_stack-> [ display ]
                    -with_list-> [              ]
                    -use_list-> [               ]
                            ]
        -exception_list-> LIST( <exception_name> )
        -governor->   processing_unit
        -dependent_task_list-> LIST( dependent_task_info )
        -#dependent_tasks_not_terminated-> [ <integer> ]
        -#noisy_tasks_in_dependent_task_tree-> [ <integer> ]
        -entry_called-> [ <entry_name> ]
        -entry_list-> KEYED_LIST( <entry_name>,entry_list_node )
        -list_of_handlers->     handlers_list_node
        -ready_to_rendezvous_list-> LIST( entry_name )
        -nesting_level-> [ <integer> ]

 ]


display ::= KEYED_LIST( <nesting_level>,processing_unit )

dependent_task_info ::=
                                [ [ DEPENDENT_TASK_INFO]
                                        -processor-> processing_unit
                                        -terminated-> [ <boolean> ]
                                -tree_quiet-> [ <boolean> ]
```

```
                    ]

entry_list_node ::=
    [   [ENTRY_LIST_NODE]
          -state-> [ <state> ]
          -transfer_address-> [ instruction_pointer ]
          -queue-> QUEUE( entry_queue_node )
    ]

entry_queue_node ::= [ [ENTRY_QUEUE_NODE]
                              -processor-> processing_unit
                              -parameters-> KEYED_LIST( <integer>,arb_node )
                         ]

handlers_list_node ::=
    [     [HANDLERS_LIST_NODE]
       -in_handler-> [ boolean ]
       -list-> KEYED_LIST( <exception_name>,instruction_pointer )

    ]

processing_unit ::=
    [ [PROCESSING_UNIT]
        -network_node-> [ <node_id> ]
        -tar->    [ task_activation_record ]
    ]
```

```
elaboration_data ::= [ [ELABORATION_DATA]
                              -task_activation_data-> task_list_node
                     -allocator_execution_data-> task_list_node
                    ]

task_list_node ::= [ [TASK_LIST_NODE]
                              -#_non_allocated_tasks-> [ <integer> ]
                     -#_non_active_tasks-> [ <integer> ]
                     -list-> KEYED_LIST( <task_name>.task_info )
                   ]

task_info ::= [ [TASK_INFO]
                      -name-> full_id
                -processor-> [ processing_unit ]
                      -allocation_completed-> [ <boolean> ]
                -activation_completed-> [ <boolean> ]
                      -load_module-> [ load_module ]
               ]


subprogram_activation_record ::=
            [ [SAR]
                 -id-> full_id
              -context-> display
                 -user_data-> user_data
              -body-> subprogram_body
              -return_address-> instruction_pointer
```

```
            -dependent_task_list-> LIST( dependent_task_info )
            -#_dependent_tasks_not_terminated-> [ <integer> ]
            -task_activation_data-> task_list_node
            -allocator_execution_data-> task_list_node
        ]

load_module ::=
        [ [LOAD_MODULE]
            -module_id-> full_id
            -entries-> LIST( entry_node )
            -body-> task_body
            -context_of_body-> display
            -governor-> processing_unit
            -activator-> processing_unit
        ]

entry_node ::= [ [ENTRY_NODE]
                -name-> full_ident
                        -range-> range
                -formal_params-> formal_part
              ]

range ::=   [ [RANGE]
                -low-> [ <integer> ]
                -high-> [ <integer> ]
             ]
```

```
full_id ::=      [ [FULL_IDENT]   ]
                          -id-> [ <identifier> ]
                          -level-> [ <nesting_level> ]
                      ]


body ::=                        subprogram_body
                                  |task_body


subprogram_specification ::=       [ [SUBPROGRAM_SPECIFICATION]
                                  -id-> [ <identifier> ]
                                                -level-> [ curr_level ]
                                  -params-> formal_part
                                  ]


formal_part ::= LIST({parameter_specification} )


parameter_specification ::=      [ [PARAMETER_SPECIFICATION]
                                 -id_list-> identifier_list
                                                -level-> [ <integer> ]
                                 -mode-> mode
                                 -type-> type_mark
                                 -value-> code
                                ]
```

```
mode ::=   [IN] | [IN  OUT] | [OUT]

subprogram_body ::=      [ [SUBPROGRAM_BODY]
                         -specifications-> subprogram_specification
                         -declarations-> declarative_part
                         -statements-> code
                         -exceptions-> LIST( (exception_handler) )
                        ]


task_body ::= [ [TASK_BODY]
                -name-> full_id
                -declarations-> declarative_part
                -statements-> code
                -exceptions-> LIST( exception_handler )
              ]


exception_handler ::= [ [EXCEPTION_HANDLER]
                        -name_list-> LIST( exception_choice )
                        -handlers_code-> code
                        ]


exception_choice ::= [ exception_name ] | [ OTHERS ]

QUEUE(X) ::=
```

```
    [ [QUEUE]
        -first-> [ QUEUE_ELEMENT( X ) ]
        -last->  [ QUEUE_ELEMENT( X ) ]
    ]

QUEUE_ELEMENT( X ) ::= [ # ]    |
    [ [QUEUE_ELEMENT]
        -head->    [ X ]
        -rest->    [ QUEUE_ELEMENT( X ) ]
    ]


KEYED_LIST( <KEY>.MEMBER ) ::=
    [ # ] | [ [KEYED_LIST]
                -<KEY>-> MEMBER
              { -<KEY>-> MEMBER }
            ]


LIST( MEMBER ) ::= [#] | [ [LIST]
                                { --> MEMBER  }
                                  --> [#]
                           ]


STACK( KIND ) ::=  [#] | [ [STACK]
                                -head-> [ KIND ]
                                -tail-> [ STACK( KIND ) ]
                           ]
```

<node_id> ::= <identifier>

<name> ::= <identifier>

<kproc_name> ::= <identifier>

<entry_name> ::= <identifier>

<transform_id> ::= <identifier>

<function_id> ::= <identifier>

<timer_status> ::= ON | OFF

<time> ::= <integer>

<boolean> ::= TRUE | FALSE

B.

```
transform [COMPILE]
            —> *ADA_TEXT: in [ <subprogram_body> ]
            —> *MAIN_BODY: out subprogram_body

var

*COMPILE_TIME_INFO: compile_time_info := 0 [#]


compile_time_info ::= [ [COMPILE_TIME_INFO] ]

                          -level-> [ <nesting_level> ]

                          -context-> context

                      ]


context ::= KEYED_LIST( { 1 .. n},name_table )


name_table ::= LIST( name_table_item )


name_table_item ::= [ [NAME_TABLE_ITEM] ]

                        -id-> full_id

                        -type_name-> type_name

                        -declaration-> [ basic_declaration ]

                    ]



full_id ::= [ [FULL_ID]  ]

                    -id-> [ <identifier> ]

                    -level-> [ <nesting_level> ]

                ]


<nesting_level> ::= <integer>


type_mark           -- see productions
basic_declaration   --  in the pair grammar
```

```
KEYED_LIST( { <KEY> },MEMBER ) ::=

              [#] |  [ [KEYED_LIST]

                          -<KEY>-> MEMBER

                        { -<KEY>-> MEMBER }

                       ]


LIST( MEMBER ) ::= [#] |  [ [LIST]

                               { --> MEMBER }

                                 --> [#]

                             ]

-- curr_level represents an integer with value *COMPILE_TIME_INFO/.level'

begin

   parse

     *ADA_TEXT:[ <subprogram_body> ]

   generate

     *MAIN_BODY: subprogram_body#1

     *COMPILE_TIME_INFO: subprogram_body#2

pair grammar

basic_declaration ::=

                    object_declaration

                    | type_declaration

                    | subprogram_declaration

                    |task_declaration

                    |exception_declaration
```

```
basic_declaration ::=
                    object_declaration
                    | type_declaration
                    | subprogram_declaration
                    |task_declaration
                    |exception_declaration

          ;

object_declaration ::=
                   identifier_list : [constant] subtype_indication [:= expression]

object_declaration ::= #1 *a:[ [OBJECT_DECLARATION]
                     -id_list-› identifier_list
                     -type-› subtype_indication
                     -level-› [ curr_level ]
                     -value-› expression
                                -allocated-› [ ‹boolean› ]
                     ]
                         #2 add_name_to_name_table(curr_level,identifier_list,subtype_indicatio

          ;

identifier_list ::= identifier {, identifier }

identifier_list ::= LIST( s:{[ ‹identifier› ]} )
```

```
        s = { identifiers in RHS of LHS production }

        ;

type_declaration ::=
                type identifier [discriminant_part] is type_definition

type_declaration ::= #2 *a: type_definition

               add_name_to_name_table(curr_level,identifier,type_definition,[*a])

        ;

type_definition ::= access_type_definition

type_definition ::= access_type_definition

        ;

subtype_indication ::= type_mark [constraint]

subtype_indication ::= [ [SUBTYPE_INDICATION]
                         -type_info-> type_mark
                         -constraint-> constraint
                       ]

        ;
```

```
type_mark ::= type_name | subtype_name

type_mark ::= type_name | subtype_name

                    if type_name = pdtype then
                    type_mark ::= [ [PREDEFINED]
                                    -pdtype-> [ <pdtype> ]
                                ]
                    else
                    *tn := LOCATE( type_name )
                    type_mark ::= *tn/type_info
                    endif
                   ;

access_type_definition ::= access subtype_indication

access_type_definition ::=  [ [ACCESS_TYPE_DEFINITION]
                      -type_info-> type_mark
                      -constraint-> constraint
                              -defining_unit-> curr_unit
                   ]
                      ;

declarative_part ::= {basic_declarative_item}{later_declarative_item}
```

```
declarative_part ::= [ [DECLARATIVE_PART]
                       -basic_items-> LIST({ basic_declarative_item })
                       -later_items-> LIST({ later_declarative_item })
                     ]

                          | [ [#] ]

                     ;

basic_declarative_item ::= basic_declaration

basic_declarative_item ::= basic_declaration


                     ;

later_declarative_item ::=
                               body
                               |subprogram_declaration
                               |task_declaration

later_declarative_item ::=
                               body
                               |subprogram_declaration
                               |task_declaration


                     ;
body ::=                       subprogram_body
```

```
                        |task_body

body ::=                subprogram_body
                        |task_body

                ;


name ::=
                        simple_name
                        |indexed_component
                        |selected_component


name ::=
                        full_id
                        |indexed_component
                        |selected_component

                ;

task_simple_name_1 ::= simple_name

task_simple_name_1 ::= full_id

                        ;

task_simple_name_2 ::= simple_name
```

```
task_simple_name_2 ::= #1 full_id

                       #2 curr_level := curr_level + 1

                          add_entry_info_to_name_table(curr_level,full_id)

                                -- add entry names and parameter specifications
                                -- from the task specification to the
name_table

                        ;



simple_name ::=         identifier


full_id ::= [ [FULL_ID]

            -id-> [ <identifier> ]

            -level-> [ n ]          -- n is the level found by searching
                                    -- the surrounding contexts for the
         ]                          -- identifier.


         ;



indexed_component ::= name( expression  {, expression } )


indexed_component ::= [ [INDEXED_COMPONENT]

                 -name-> name

                 -indices-> KEYED_LIST({<integer>},expression )

                ]


                ;
```

```
selected_component ::= name.selector

selected_component ::= [ [SELECTED_COMPONENT]
                         -name-> name
                         -selector-> selector
                        ]

                        ;


selector ::=                simple_name

                            |all


selector ::=                full_id
                      | [ALL]

                        ;


allocator ::= new type_mark

allocator ::= [ [ALLOCATOR] -->
                  [ REP( type_mark ) ] &t -->
                  [ ALLOC( &t ) ] &ptr -->
                        [#]
                ]

                    ;
```

```
sequence_of_statements ::= statement { statemant }

sequence_of_statements ::= LIST( { statement } )

                ;

statement ::=           simple_statement
                        |compound_statement

statement ::=           simple_statement
                        |compound_statement

                ;

simple_statement ::= null_statement
                     |assignment_statement
                     |delay_statement
                     |raise_statement
                     |procedure_call_statement
                     |return_statement
                     |entry_call_statement
                     |abort_statement

simple_statement ::= null_statement
                     |assignment_statement
                     |delay_statement
```

```
                    |raise_statement
                    |procedure_call_statement
                    |return_statement
                    |entry_call_statement
                    |abort_statement

                  ;

compound_statement ::=      accept_statement
                           |select_statement

compound_statement ::=      accept_statement
                           |select_statement

                  ;

null_statement ::= null;

null_statement ::= [ [NULL_STATEMENT] -->
                   [ NOOP ] -->
                   [#]
                 ]

                  ;
```

```
assignment_statement ::= variable_name := expression;

assignment_statement ::= [ [ASSIGNMENT_STATEMENT] -->
                           [ REF( variable_name ) ] &z -->
                           expression  &e -->
                           [ ASSIGN( &z,&e ) ] -->
                           [#]
                         ]

                         ;

return_statement ::= return [expression];

return_statement ::= [ [RETURN_STATEMENT] -->
                       expression &e -->
                       [ RETURN( &e ) ] -->
                       --> [#]
                     ]

                       ;

subprogram_declaration ::=   subprogram_specification;

subprogram_declaration ::=   subprogram_specification;

                         ;
```

```
subprogram_specification_1 ::= procedure identifier [formal_part]

subprogram_specification_1 ::= #1 *a:[ [SUBPROGRAM_SPECIFICATION]
                        -id-> [ <identifier> ]
                        -level-> [ curr_level ]
                        -params-> formal_part
                        ]
                      #2 add_name_to_name_list(curr_level,identifier,subprogram,[*a])
                    ;

subprogram_specification_2 ::= #1 [ [SUBPROGRAM_SPECIFICATION]
                        -id-> [ <identifier> ]
                                  -level-> [ curr_level ]
                        -params-> formal_part
                        ]

                       #2 curr_level := curr_level + 1

                                ;

formal_part ::= (parameter_specification {; parameter_specification}

formal_part ::= LIST( (parameter_specification} )

                ;

parameter_specification ::= identifier_list : mode type_mark [:= expression]
```

```
parameter_specification ::= #1 *a:[ [PARAMETER_SPECIFICATION]
                            -id_list-> identifier_list
                            -level-> [curr_level + 1]
                            -mode-> mode
                            -type-> type_mark
                            -value-> expression
                          ]
                      #2 add_name_to_name_list(curr_level+1,identifier_list,[*a])
                      ;

mode ::= [in] | in out | out

mode ::= [IN] | [IN OUT] | [OUT]
                       ;

subprogram_body ::=
            subprogram_specification_2 is
             [declarative_part]
             begin
                sequence_of_statements
               [exception
                    exception_handler
                    {exception_handler}  ]
             end [designator];
```

```
subprogram_body ::= #1 [ [SUBPROGRAM_BODY]
                    -specifications-> subprogram_specifications
                    -declarations-> declarative_part
                    -statements-> [ prelude -->
                                    overture -->
                                    sequence_of_statements -->
                                    epilog -->
                                    [#]
                                     ]
                    -exceptions-> LIST( {exception_handler} )
                  ]
            #2 curr_level := curr_level - 1
                  ;
```

```
procedure_call_statement ::= procedure_name [ actual_parameter_part];

procedure_call_statement ::= [ [PROCEDURE_CALL_STATEMENT]
                               actual_parameter_part &params -->
                               [ REF(procedure_name ) ] &p -->
                               [ CALL( &p,&params ) ] -->
                               [#]
                              ]
```

```
                    ;

actual_parameter_part ::= (parameter_association {, parameter_association})

actual_parameter_part ::= LIST( parameter_association )

                    ;

parameter_association ::= [formal_parameter =>] actual_parameter

parameter_association ::= [ [PARAMETER_ASSOCIATION] -->
                    actual_parameter -->
                    [add_to_parameter_list(param_info,param_id,mode,type)] -->    -- param_id
filled in from
          -- corresponding formal parameter
                    [#]
               ]

               ;


formal_parameter ::= parameter_simple_name

formal_parameter ::= parameter_simple_name

                  ;


actual_parameter ::= expression
                    |variable_name
```

```
                    |type_mark (variable_name)

actual_parameter ::= [ [VAL] -->
                     expression &e -->
                           [fill_in_param_info(&e,'VALUE',null)] -->
                     [#]
                   ]

                   | [ [REF] -->
                    [ REF( variable_name ) ] &n  -->
                           [fill_in_param_info(&n,'ADDR',null)] -->
                    [#]
                     ]

                   | [ [REFT] -->
                    [REF(variable_name)] &n -->
                    [REF(type_mark) ] &t -->
                           [fill_in_param_info(&n,'TADDR',&t)] -->
                           [#]
                     ]


                       ;


task_declaration ::= task_specification;

task_declaration ::= task_specification;
```

```
        [ declarative_part ]
      begin
           sequence_of_statements
      [exception
           exception_handler
           {exception_handler}
      end [task_simple_name];


task_body ::= #2 *body: [ [TASK_BODY]
                        -name-> task_simple_name_2
                        -declarations-> declarative_part
                        -statements-> [prologue -->
                                       overture -->
                                  sequence_of_statements -->
                                  epilog -->
                                  [#] ]
                        -exceptions-> LIST( exception_handler )
                       ]

           find_in_name_list( task_simple_name_2,*n )
           *n/body' := task_body'
        curr_level := curr_level - 1


                        ;


entry_declaration ::=
```

```
                    ;

task_specification ::=
        task [type] identifier [is
           {entry_declaration}
           {representation_clause}
        end [task_simple_name]]


load_module_template ::= #2 *a:[ [LOAD_MODULE_TEMPLATE]
                                  -module_id-> [ [FULL_ID]
                                      -id-> [ <identifier> ]
                                      -level-> [ curr_level ]
                                                ]
                      -entries-> LIST( {entry_declaration} )
                                  -body-> [#]
                                  -context_of_body-> [#]
                                  -governor-> [#]
                                  -activator-> [#]
                      ]

                   add_name_to_name_list(curr_level,identifier,task,[*a])


                   ;

task_body ::=
      task body task_simple_name_2 is
```

```
   entry identifier [(discrete_range)] [formal_part];

entry_declaration ::= #1 [ [ENTRY_DECLARATION]
                        -id-> identifier
                        -level-> [ curr_level ]
                        -range-> range
                        -formal_part-> formal_part
                       ]
                    #2 add_name_to_name_list(curr_level,identifier,entry )
                    ;

entry_call_statement_1 ::= entry_name[actual_parameter_part];

entry_call_statement_1 ::= [ [ENTRY_CALL] -->
                         [REF(entry_name) ]&E,&pssr -->
                         actual_param_part &params -->
                         [ entry_call_proc( &pssr,&E,&params ) ] -->
                         [#]
                       ]
                    ;

entry_call_statement_2 ::= entry_name[actual_parameter_part];
```

```
entry_call_statement_2 ::= [ [ENTRY_CALL] -->
                           [REF(entry_name) ]&E,&pssr -->
                           actual_param_part &params -->
                           [#]
                         ]

                       ;


accept_statement_1 ::=

    accept entry_simple_name [(expression)] [formal_part] [ do
         sequence_of_statements
    end [entry_simple_name] ];

accept_statement_1::=  [ [ACCEPT_STATEMENT] -->
                       [REF(entry_simple_name) ] &n -->
                       expression &e -->
                       [REF(formal_part)] &f -->
                       [ accept_proc(&e,&f,&n) ] -->
                       sequence_of_statements -->
                       [ end_of_rendezvous ] -->
                       [#]
                      ]


                      ;
```

```
accept_statement_2 ::= accept_part_1 accept_part_2

accept_part_1 ::= entry_simple_name [ (expression) ] [formal_part]

accept_part_2 ::= [ do sequence_of_statements end [entry_simple_name] ];
```

```
accept_statement_2 ::= accept_part_1 accept_part_2

accept_part_1::=     [ [ACCEPT_PART_1] -->

                [REP(entry_simple_name) ] &n -->

                expression &e -->

                formal_part &f -->

                        [#]

                     ]
```

```
accept_part_2::=    [ [ACCEPT_PART_2] -->

                [ accept_proc(&e,&f,&n) ] -->

                sequence_of_statements -->

                [ end_of_rendezvous ] -->

                [#]

                   ]

                   ;
```

```
delay_statement_1 ::= delay simple_expression;

delay_statement_1 ::= [ [DELAY_STATEMENT_1] -->

               simple_expression &d -->

               [ set_timer ( &d,*a ) ] -->

               [ state_becomes( ' suspended:at delay ' ) ] -->
```

```
                *a:[#]
                 ]

                 ;

delay_statement_2 ::= delay simple_expression;

delay_statement_2 ::= [ [DELAY_STATEMENT_2] -->
                simple_expression &d -->
                [#]
                 ]

                 ;

select_statement ::= selective_wait
                    |conditional_entry_call
                    |timed_entry_call

select_statement ::= selective_wait
                    |conditional_entry_call
                    |timed_entry_call

                      ;

selective_wait ::=
      select
```

```
        select_alternative
        {or
          select_alternative}
        else
           sequence_of_statements
    end select;

selective_wait ::= [ [SELECTIVE_WAIT_STATEMENT] -->
                     [ set_up_temp_data_str ] -->
                     LIST({ select_alternative}) -->
                     [ [BRANCH]
                -condition-> [check_if_any_open_guard]
                -alternatives-> [ [KEYED_LIST]
                             -true-> [ perform_select ]
                             -false-> sequence_of_statements
                           ]
               ] -->
               [ release_temp_data_str ] -->
               **end_of_select:[#]
        ]

        ;

selective_wait ::=
    select
        select_alternative
```

```
        {or
          select_alternative}
      end select;

selective_wait ::= [ [SELECTIVE_WAIT_STATEMENT] -->
                     [ set_up_temp_data_str ] -->
                     LIST({ select_alternative}) -->
                     [ [BRANCH]
                -condition-> [check_if_any_open_guard]
                -alternatives-> [ [KEYED_LIST]
                             -true-> [ perform_select ]
                             -false-> [ RAISE_EXCEPTION('SELECT ERROR' ) ]
                           ]
               ] -->
               [ release_temp_data_str ] -->
               **end_of_select:[#]
          ]

          ;

select_alternative  ::= [ when condition => ]
                          selective_wait_alternative

select_alternative ::= [ [SELECT_ALTERNATIVE] -->
                [ [BRANCH]
                    -condition-> condition
                    -alternatives-> [ [KEYED_LIST]
```

```
                        -true-> selective_wait_alternative
                        -false-> [#]
                         ]
                ] -->
                        [#]
                ]

                ;

selective_wait_alternative ::= accept_alternative
                             |delay_alternative
                             |terminate_alternative

selective_wait_alternative ::= accept_alternative
                             |delay_alternative
                             |terminate_alternative

                    ;

accept_alternative ::= accept_statement_2 [sequence_of_statements]
accept_statement_2 ::= accept_part_1 accept_part_2

accept_alternative ::= [ [ACCEPT_ALTERNATIVE] -->
                  accept_ part_1 &n &e &f -->
                          *a
```

```
                    *b: accept_part_2 -->
                sequence_of_statements -->
                **end_of_select
                *a:[put_on_open_guards_list( &n,&e,&f,*b ) -->
                [#]
               ]


               ;


delay_alternative_1 ::= delay_statement_2 [sequence_of_statements]


delay_alternative_1 ::= [ [DELAY_ALTERNATIVE] -->
               delay_statement_2 &d -->
                        *a
               *c:sequence_of_statements -->
               **end_of_select
               *a:[ update_smallest_open_delay( &d,*c ) ] -->
               [#]
                ]


              ;


terminate_alternative ::= terminate


terminate_alternative ::= [ ['TERMINATE_ALTERNATIVE] -->
```

```
                [ set_open_terminate_flag ] -->
                [#]
              ]

              ;

conditional_entry_call ::=
       select
          entry_call_statement_2
          [sequence_of_statements_1]
        else
          sequence_of_statements_2
        end select;

conditional_entry_call ::= [ [CONDITIONAL_ENTRY_CALL] -->
                  entry_call_statement_2  &E,&pssr,&params -->
                  [ request_rendezvous (&E,&pssr,$params) ] -->
                  [ [BRANCH]
                  -condition-> [ rendezvous_possible( ) ] -->
                  -alternatives-> [ [KEYED_LIST]
                              -true-> sequence_of_statements_1
                              -false-> sequence_of_statements_2
                            ]
                  ] -->
                  [#]
              ]
```

```
                ;

timed_entry_call ::=
        select
           entry_call_statement
           [sequence_of_statements_1]
        or
            delay_statement_2
            [ sequence_of_statements_2 ]
        end select;

timed_entry_call ::= [ [TIMED_ENTRY_CALL] -->
                  delay_statement_2 &d  -->
                        *b
                        *a:sequence_of_statements_2 -->
                  *end
                        *b:[ set_timer( &d,&a ) ] -->
                  entry_call_statement_1 -->
                  sequence_of_statements_1 -->
                  *end:[#]
                ]

                ;


abort_statement ::= abort task_name {,task_name};
```

```
abort_statement ::= LIST( {ABORT_TASK(task_name) })

ABORT_TASK(X) ::= [ [ABORT] —›
                    [REP(X) ] &n —›
                    [ abort_exec( &n ) ] —›
                    [#]
                  ]

                  ;
```

```
exception_declaration ::= identifier_list : exception;

exception_declaration ::= [ [EXCEPTION_DECLARATION]
                            -id_list-› identifier_list
                            -type-› [EXCEPTION]
                          ]

                          ;
```

```
exception_handler ::=
   when exception_choice {|exception_choice } =›
        sequence_of_statements
```

```
exception_handler ::= [ [EXCEPTION_HANDLER]
                        -name_list-› LIST( exception_choice )
                        -handlers_code-› sequence_of_statements
```

```
                ]

                ;

exception_choice ::= exception_name

                    |others

exception_choice ::= [ exception_name ] | [ OTHERS ]

               ;


raise_statement ::= raise [exception_name];

raise_statement ::= [ [RAISE_STATEMENT] -->
                [REP( exception_name ) &n -->
                [ raise_exception ( &n ) ] -->
                [#]
               ]

end COMPILE
```

C.

```
procedure  FIRST is
  task type SIMPLE is
    entry X( I: in integer );
  end SIMPLE;

  task type COMPUTE is
    entry Y ( I: out integer );
  end COMPUTE;

  S:SIMPLE;
  C:COMPUTE;
  I:integer :=6;

  task body SIMPLE is
    A:integer :=10;
    B:integer;
    begin
      accept X (I:in integer);
         B := A + I;
      end X;
      print(B);
    end SIMPLE;

  task body COMPUTE is
    C:integer :=3;
    begin
      S.X(C);
```

```
    accept Y(I:out integer);
      I := C + 2;
    end Y;
  end COMPUTE;

begin  -- FIRST
  C.Y(I);
  I := I + 5;
  PRINT(I);
end FIRST;
```

D.

```
[ [SUBPROGRAM_BODY]
   -specifications-> [ [SUBPROGRAM_SPECIFICATION]
                        -id-> [FIRST]
                        -level-> [0]
                        -params-> [#]
                     ]

   -declarations-> [ [LIST] -->
               [ [OBJECT_DECLARATION]
                  -id_list-> [ [LIST] --> [S] --> [#] ]
                  -type-> [ [SUBTYPE_INDICATION]
                                 -type_info-> *lmtSIMPLE
                                 -constraint-> [#]
                           ]
                  -level-> [1]
                  -value-> [#]
               ] -->

               [ [OBJECT_DECLARATION]
                  -id_list-> [ [LIST] --> [C] --> [#] ]
                  -type-> [ [SUBTYPE_INDICATION]
                                 -type_info-> *ltmCOMPUTE
                                 -constraint-> [#]
                           ]
                  -level-> [1]
                  -value-> [#]
```

```
        ] —>

              [ [OBJECT_DECLARATION]
            -id_list-> [ [LIST] —> [I] —> [#] ]
            -type-> [ [PREDEFINED]
                                -pdtype-> [INTEGER]
                            ]
            -level-> [1]
            -value-> [#]
        ] —>

          [#]
    ]

-statements-> [ [LIST] —>
        prologue —>
        overture —>
              [ [ENTRY_CALL] —>
          [ REF( C,1),(Y,1)]  &pssr,&E —>
          [REF( I,1 ) ] &params —>
          [entry_call_proc(&pssr,&E,&params)] —>
          [#]
              ] —>

        [ [ASSIGNMENT_STATEMENT] —>
          [ REF( (I,1) ) ] &addr —>
          [ [EXPRESSION] —>
```

```
                [REF( (I,1) ) ] &a -->
                [REF( 5 )     ] &b -->
                [ADD( &a,&b )] &c -->
                [#]
              ] -->
              [ ASSIGN( &addr,&c ) ] -->
              [#]
            ] -->
            [REF( I,1 ) ] &out -->
            [PRINT(&out) ] -->
            epilog -->
            [#]
            ]

   -exceptions-> [#]
]

*lmtSIMPLE:          [ [LOAD_MODULE_TEMPLATE]
                        -module_id-> [ [FULL_ID ]
                                       -id-> [SIMPLE]
                                       -level-> [1]
                                     ]
                     -entries-> [ [LIST] -->
                                  [ [ENTRY_DECLARATION]
                                  -id-> [X]
                                  -level-> [2]
```

```
                    -range-> [#]
                    -formal_part-> [ [LIST] -->
                              [ [PARAMETER_SPECIFICATION]
                                 -id_list-> [ [LIST] --> [I] --> [#] ]
                                 -level-> [2]
                                 -mode-> [IN]
                                 -type-> [ [PREDEFINED]
                                                              -pdtype-> [INTEGER]
                                                          ]

                                 -value->[#]
                              ] -->
                              [#]
                               ]
                    ] -->
                    [#]
                     -body-> *task_body_simple
                     -context_of_body-> [#]
                     -governor-> [#]
                     -activator-> [#]
               ]

*lmtCOMPUTE:        [ [LOAD_MODULE_TEMPLATE]
                     -module_id-> [ [FULL_ID ]
                                   -id-> [COMPUTE]
                                   -level-> [1]
                                  ]
```

```
-entries-> [ [LIST] -->
           [ [ENTRY_DECLARATION]
           -id-> [Y]
           -level-> [2]
           -range-> [#]
           -formal_part-> [ [LIST] -->
                     [ [PARAMETER_SPECIFICATION]
                        -id_list-> [ [LIST] --> [I] --> [#] ]
                        -level-> [2]
                        -mode-> [OUT]
                        -type-> [ [PREDEFINED]
                                                    -pdtype-> [INTEGER]
                                                ]

                        -value->[#]
                     ] -->
                     [#]
                      ]
          ] -->
          [#]
           -body-> *task_body_compute
           -context_of_body-> [#]
           -governor-> [#]
           -activator-> [#]
      ]

*task_body_simple:[ [TASK_BODY]
```

```
 -name-> [ [FULL_ID]
-id-> [S]
-level-> [2]
]
 -declarations-> [ [LIST] -->
                   [ [OBJECT_DECLARATION]
                  -id_list-> [ [LIST] --> [A] --> [#] ]
                  -type-> [integer]
                  -level-> [2]
                  -value-> [10]
                ] -->

                   [ [OBJECT_DECLARATION]
                  -id_list-> [ [LIST] --> [B] --> [#] ]
                  -type-> [ [PREDEFINED]
                                  -pdtype-> [INTEGER]
                               ]

                  -level-> [2]
                  -value-> [#]
                ] -->
                     [#]
              ]

-statements-> [  [LIST] -->
          prologue-->
          overture -->
```

```
                [ [ACCEPT_STATEMENT] -->
               [REF( X,1 ) ] &n -->
               [REF( I,2 ) ] &f -->
               [accept_proc(&f,&n)] -->
                  [ [ASSIGNMENT_STATEMENT] -->
                 [REF( B,2 ) ] &addr -->
                 [ [EXPRESSION] -->
                   [REF( A,2 ) ] &op1 -->
                   [REF( I,2 ) ] &op2 -->
                   [ADD( op1,op2 ) ] &e -->
                   [#]
                                ] -->
                    [ASSIGN( &addr,&e ) ] -->
                    [#]
                 ] -->
               [end_of_rendezvous] -->
               [#]
                 ] -->

                            [REF ( B,2 ) ] &out -->
                 [PRINT( &out ) ] -->
                 [#]
          ]

     -exceptions->[#]
]
```

```
*task_body_compute:[ [TASK_BODY]
                     -name-> [ [FULL_ID]
                    -id-> [C]
                    -level-> [2]
                    ]
                     -declarations-> [ [LIST] -->
                                     [ [OBJECT_DECLARATION]
                                  -id_list-> [ [LIST] --> [C] --> [#] ]
                                  -type-> [ [PREDEFINED]
                                                  -pdtype-> [INTEGER]
                                           ]

                                  -level-> [2]
                                  -value-> [3]
                              ] -->
                                    [#]
                              ]


                -statements-> [ [LIST] -->
                           prologue-->
                           overture -->
                                       [ [ENTRY_CALL] -->
                           [REF( S,1),(X,1)]  &pssr,&E -->
                           [REF( C,2 ) ]&params -->
                           [entry_call_proc(&pssr,&E,&params)] -->
                           [#]
                                        ] -->
```

```
        [ [ACCEPT_STATEMENT] -->
        [REF( Y,1 ) ] &n -->
        [REF( I,2 ) ] &f -->
        [accept_proc(&f,&n)] -->
          [ [ASSIGNMENT_STATEMENT] -->
          [REF( I,2 ) ] &addr -->
          [ [EXPRESSION] -->
            [REF( C,2 ) ] &op1 -->
            [REF( 2 ) ] &op2 -->
            [ ADD( op1,op2 ) ] &e -->
            [#]
                          ] -->
          [ASSIGN( &addr,&e ) ] -->
          [#]
          ] -->
        [end_of_rendezvous] -->
        [#]
          ] -->
          epilog -->
          [#]
      ]

    -exceptions-> [#]

]
```

# Appendix 2

## Questions on the Ada '82 Language Reference Manual.

Questions and comments about wrong, confusing, unclear, and incomplete parts in the tasking sections (mainly chapter 9 but some of chapter 11) of intra-canvass Ada Reference Manual.

(1) General

The examples appearing in the 1982 reference manual are no more than those in the 1980 version. Most are examples of proper statement syntax only. There are places in the manual where even a simple example would clarify more than the volume of text. The reference manual needs more examples; at least one for each language feature. Where an example refers to or uses a previous example, an explicit reference should be given.

(2) Chapter 9, page 1, paragraph 1, line 1

It is stated that the execution of a program which contains no task procedes according to the rules described by the manual less chapter 9. In a multiprogrammed system, main programs look remarkably like tasks executing independently. Is the main program a task (with no entries) or not?

(3) Chapter 9, page 1, paragraph 1, line 4

"The effect of ... a program is defined in terms of a sequential execution of its actions in some order..." What does "some order" mean? We realize that the intended order is that traditionally found in implementations of Algol-descended languages with rearrangements and optimizations restricted as in chapter 11. However, the manual does not specify that, it says "some order."

(4) Chapter 9, page 1, paragraph 2, line 4

We know that two tasks are synchronized at the beginning and end of a rendezvous, and that tasks are synchronized with their declaring parent at their activation, but are there other places? For instance, are tasks "synchronized" during execution of an ABORT statement since in that case they are not operating independently. This is the first occurrence of the term "synchronize." It is a technical term in the definition of Ada semantics. It must be precisely defined.

(5) Chapter 9, page 1, paragraph 4, line 3

There are three kinds of program units of which programs can be composed according to chapter 9 but four according to chapter 7.

(6) Chapter 9, page 1, paragraph 4, line 3

What is the intent of a program unit (the term is not defined)? If I write a program unit which consists solely of a task unit or generic unit, what can I do with it? (Dare we ask "What is a main program?")

(7) Chapter 9, page 2 section 9.1 paragraph 3

"task [type] identifier [is ... end [simple_name]]" What is the distinction that is being made between an identifier and a simple_name? Is the reference manual alluding to the symbol table operation and to the relationship between the lexical analyzer and parser of a particular compiler?

(8) Chapter 9, page 2 section 9.1 paragraph 5
"...the body can ... be used for the execution of tasks designated by objects of the ... task type." From reading descriptions elsewhere in the manual it seems to us that the term "values of objects" would be more appropriate here. The continuation of the fiction that a task object and its value are somehow different when we are told that tasks behave as constants, seems silly. It does provide consistency with the descriptions of other kinds of objects and their values but can add confusing verbiage to an already confusing chapter (besides, it contributed to this mistake in the manual itself).

(9) Chapter 9, page 3 section 9.1 paragraph 1
We had an argument about when or whether it would be legal for a task to refer to itself, especially by its type identifier. We eventually came up with several valid cases, but the point here is that the manual slips the capability in and certainly does not expand on it. The material explaining how a task type name serves as a task name is very cryptic and could do with some elaboration. A separate notation for self reference would be nice.

(10) Chapter 9, page 4 section 9.2 paragraph 1
We were under the impression (and the first note in the designated section seems to support this) that task types could be passed as generic actual parameters at instantiations of generic units, yet this section, besides the note, ignores such usage. Was it ever decided whether omission in the reference manual constituted a prohibition?

(11) Chapter 9, page 5 section 9.2 note 1
The business of modes allowed and disallowed for generic parameters whose types are task types is very confusing. This note needs elaboration or it needs to be moved to an appropriate place in the chapter on generics. Why are tasks not allowed as actual parameters corresponding to generic formal parameters with mode IN since tasks by definition are "constant"?

(12) Chapter 9, section 9.3
The manual is very explicit about when task objects declared in a declarative part get activated (not before and not after the following BEGIN), and about when task objects created via an allocator get activated. When do task objects created via an allocator in the initialization of an object of an access type in a declarative part get activated? This is important in terms of understanding what is completed and what is terminated should an exception occur during the activation of one of these tasks. We do not understand when these tasks get activated! We are also concerned about the apparent inconsistency in the fates of declared and allocated tasks which experience exceptions during their activations.

(13) Chapter 9 section 9.3
In 9.3 activation is defined to be the elaboration of the declarative part of a task body. When does a task proceed after its activation has been completed? NOTE: In July 1980 Ada, section 9.3 states, ' Each task can continue its execution as a parallel entity once its activation is

completed.' Why was this omitted?

(14) Chapter 9, page 21 section 9.11
Why is task synchronization between activator and activatee only mentioned in shared variables? We need a definition of synchronization.

(15) Chapter 9, page 7 section 9.3
References at the end of section 9.3 (and probably elsewhere) still have "?" in them. Will they be replaced?

(16) Chapter 9, page 6 section 9.3 paragraphs 1 & 4
When tasks are being activated after a begin, if the activation raises an exception the task becomes completed. On the other hand paragraph 4 states that if a task has been created by the execution of an allocator and an exception is raised during its activation then the task becomes terminated. Are the cases really different and if so why? NOTE: Ch 11 p8 s11.4.2(d) says that the task would be completed in both cases.

(17) Chapter 9, page 6 section 9.3 paragraphs 1 & 4
"other tasks are unaffected" Does this include dependents or does it refer back to 'these tasks' - the tasks being activated? This is not clear.

(18) Chapter 9, page 8 section 9.4 [paragraph 2 ... Example]
Use of "unit" vs. "task" is extremely confusing. Text should replace 'certain unit' by 'parent unit'. The meaning here can be completely missed very easily (some of us did on first reading).

(19) Chapter 9, page 7 section 9.4 paragraph following (c)
The new definition of dependency needs further explanation. We suggest adding a note explaining that because of the definition of termination and the rules about leaving subprograms and blocks, only tasks defined in a unit or contained in an inner package need be checked for termination.

(20) Chapter 9, page 8 section 9.4 Example
Example is not clear because we don't know where G.ALL was activated. We suggest that comments should be ammended to read "await termination of G.ALL if it was ever activated no matter where."

(21) Chapter 9, page 10 section 9.5 (last paragraph before the example)
This needs to be rewritten more clearly. Chapter 11 section 11.5 contains a clear explanation of the situation which could be copied or referenced.

(22) Chapter 9, page 11 section 9.5 note 2
What if an entry has OUT parameters but the accept has no statements -- what happens if you use the parameters?

(23) Chapter 9, page 12 section 9.6
Typo in PACKAGE CALENDAR: 2009 should be 2099

(24) Chapter 9, page 13 section 9.6 note 1
Heed your note and make the correction (we would have liked to have seen this explanation).

(25) Chapter 9, page 14 section 9.7.1 paragraph 2, line 1
The line: "A selective wait must contain at least one alternative ... " should read: "A selective wait must contain at least one select alternative ... " since that is the non-terminal used in the syntactic definition.

(26) Chapter 9, page 14 section 9.7.1 paragraph 2
Parenthesized comments in this paragraph specifying the combinations of terminate, delay and else parts allowed in a select statement should not be parenthesized -- they are too important. They should be separately stated and elaborated.

(27) Chapter 9, page 15 section 9.7.1 dashed paragraph 1
When does the delay start? Is it safe for the programmer to assume that the total amount of time required for the select statement if no rendezvous is posssible, is no greater than that given in the delay statement or is the time needed to evaluate any guards not included in the execution time of a select statement?

(28) Chapter 9, page 16 section 9.7.2 paragraph 1, line 1
Use of the word immediately is confusing. A conditional entry call may take an arbitrary amount of (communication) time to execute even if no rendezvous occurs.

(29) Chapter 9, page 17 section 9.7.3
What does the delay include in a timed entry call? Is it the time on the entry queue only, or does it include "message transmission" time to/from caller from/to callee? If the latter, how do we implement this when one task is on Earth and the other on Mars (this is not a flippant question)? Also, if no scheduling algorithm is assumed by the language definition, how can the "correct" execution be guaranteed? We assume a timed entry call with delay 0 really means the programmer is prepared to wait 0 seconds on the entry queue. It is clearly impossible to have any other meaning because an entry call always takes some time. Thus we assume delay 1 means wait for a duration of 1 on the queue. Is this correct?

(30) Chapter 9, page 18 section 9.8 rule
The word "sensibly" is not appropriate in this context. It, is far too ambiguous for a document puporting to be a language definition.

(31) Chapter 9, page 18 section 9.8 last paragraph
If two tasks rendezvous, one with priority 5 and the other without defined priority, is it a valid implementation for the rendezvous to always occur with priority PRIORITY'LAST+1?

(32) Chapter 9, page 20 section 9.10
It is impossible to guarantee that a task named in an ABORT statement will not proceed beyond an accept (etc.) after the aborting task thinks

the aborted task has been marked abnormal. If the tasks were running on different physical processors the communication time for the abort message could be arbitrarily long. Is it legitimate for a task that has been marked abnormal to execute an ABORT statement? The manual implies 'yes'.

(33) Chapter 9, page 20 section 9.10
What happens to the caller/callee in a rendezvous when the callee/caller is aborted? (This is explained in Chapter 11 but should be in section 9.10 also)

(34) Chapter 9, page 21 section 9.11
This section as a whole and the usage of the SHARED_VARIABLE_UPDATE procedure in particular is not at all clear -- we need an example. Also you should point out that use of shared variables makes programs non-portable because this facility may not cover all types in an implementation.

(35) Chapter 9, page 20 section 9.10 (general)
Can the task below be terminated by an ABORT statement once the rendezvous has begun?

```
TASK TRAP IS
  ENTRY X;
END;
TASK BODY TRAP IS
  BEGIN
    ACCEPT X DO
      LOOP
        NULL;
      END LOOP;
    END;
  END;
```